МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ТУЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт прикладной математики и компьютерных наук Кафедра информационной безопасности

Разработка программ по варианту № 07

(Графы: мин. остовное дерево, Графы: нахождение предков в деревьях и наименьших общих предков на языке с++)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
по дисциплине

_____

(полное наименование учебной дисциплины)

Студент гр.	_____	_____	_____
	(индекс группы)	(подпись и дата)	(инициалы и фамилия)
Руководитель	доц. ИПМКН, к.т.н.,	_____	Сафронова М.А.
	(должность и ученая степень)	(подпись и дата)	(инициалы и фамилия)

ТУЛА 2023

УТВЕРЖДАЮ

Дир. ИПМКН

_____А.А.Сычугов

"___"_____20__г.

ЗАДАНИЕ

на курсовую работу по программированию

студента гр. _____

(ФИО, группа)

ТЕМА: _____

_____

(Название, номер варианта)

Исходные данные _____

_____

_____

Задание получил:_____

(ФИО, подпись)

Дата выдачи задания :_____

Задание выдал:_____

(ФИО, подпись)

Срок защиты курсовой работы: _____

Замечания консультанта: _____

_____

_____

К защите допущен. Консультант работы _____

"___"_____20__г.

Оглавление

Введение

В данной курсовой работе рассматриваются две задачи, связанные с графами: минимальное остовное дерево и нахождение предков в деревьях и наименьших общих предков. Графы являются важным математическим инструментом для моделирования и анализа сложных систем, а решение указанных задач имеет множество практических применений.

В задаче о минимальном остовном дереве требуется найти подмножество ребер минимальной суммарной стоимости, которые связывают все вершины графа, при этом не образуя циклов. Это позволяет найти оптимальное дерево, которое соединяет все вершины графа с минимальными затратами. Методы решения этой задачи широко применяются в областях, таких как транспортное планирование, сетевое проектирование, оптимизация маршрутов и других.

Задача о нахождении предков и наименьших общих предков в деревьях является классической задачей алгоритмики. В ней требуется найти для каждой вершины дерева ее предка и наименьшего общего предка с другой вершиной. Эта задача имеет множество практических применений, например, в семантическом анализе текстов, компьютерном зрении, генетике и других областях.

В данной работе будет представлено описание задач, входных и выходных данных, алгоритмы решения, общие требования к программе, описание структуры программы, инструкции по эксплуатации программы и контрольные примеры для каждой задачи. Также будет проведен анализ результатов и представлено заключение по выполненной работе.

1. Минимальное остовное дерево

1.1 Постановка задачи

Задача "Минимальное остовное дерево" заключается в поиске такого подмножества ребер во взвешенном связном неориентированном графе, которое содержит все вершины графа и имеет минимальную сумму весов ребер.

Формально постановка задачи выглядит следующим образом:

Дано связный неориентированный граф $G = (V, E)$, где V - множество вершин графа, E - множество ребер графа. Каждому ребру $e \in E$ сопоставлено неотрицательное вещественное число $w(e)$, называемое весом ребра.

Требуется найти подмножество ребер $T \subseteq E$ такое, что:

- T содержит все вершины графа G;
- T не содержит циклов;
- Сумма весов ребер в T минимальна.

Цель задачи заключается в построении минимального остовного дерева, которое представляет собой подграф исходного графа G, содержащий все вершины исходного графа и имеющий наименьшую сумму весов ребер.

Решение этой задачи имеет множество практических применений, таких как оптимизация сетей связи, планирование маршрутов, проектирование электрических сетей и других областях, где необходимо найти оптимальное соединение между объектами.

1.2 Описание входной и выходной информации

В задаче "Минимальное остовное дерево" входная информация представляет собой взвешенный связный неориентированный граф. Граф представляет собой совокупность вершин и ребер, где каждое ребро имеет свой вес.

Входная информация включает следующие параметры:

1. Количество вершин (n): целое положительное число, определяющее общее количество вершин в графе. Вершины обычно нумеруются от 1 до n.
2. Множество ребер (E): список ребер графа, где каждое ребро представлено парой вершин и его весом. Каждое ребро обозначается (u, v, w), где u и v - вершины, соединенные ребром, а w - вес ребра. Вес ребра может быть любым числом и отражает стоимость или длину ребра.

   Пример входных данных:

- Количество вершин (n) = 5
- Множество ребер (E) = [(1, 2, 4), (1, 3, 2), (2, 3, 1), (2, 4, 3), (3, 4, 5), (3, 5, 6), (4, 5, 7)]

Выходная информация в задаче "Минимальное остовное дерево" представляет собой минимальное остовное дерево, которое является подграфом исходного графа и содержит все его вершины.

Выходные данные представляются в формате списка ребер, где каждое ребро обозначается (u, v, w), где u и v - вершины, соединенные ребром, а w - вес ребра.

Пример выходных данных:

- Минимальное остовное дерево = [(1, 3, 2), (2, 3, 1), (2, 4, 3), (3, 5, 6)]

Выходная информация демонстрирует, какие ребра должны быть включены в минимальное остовное дерево и какие веса у этих ребер. Визуальное представление остовного дерева изображена на рисунке 1.

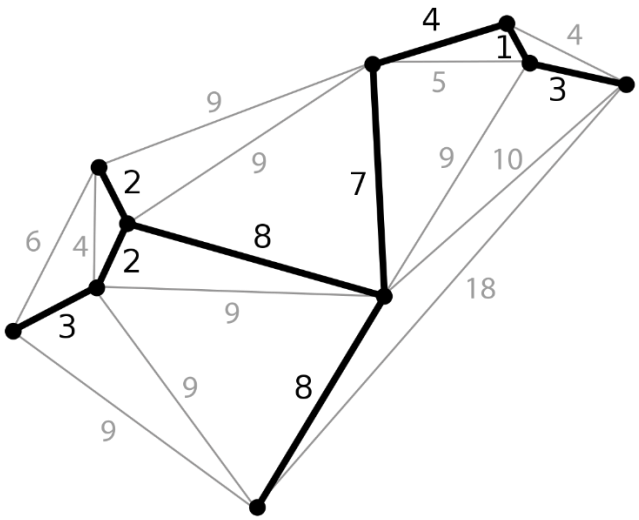

Рисунок 1 – Визуализация остовнового дерева графа.

1.3 Алгоритм решения задачи

1.3.1 Алгоритм Прима

Для решения задачи "Минимальное остовное дерево" существуют несколько известных алгоритмов. Один из наиболее распространенных алгоритмов - алгоритм Прима.

Алгоритм Прима основан на жадной стратегии и позволяет построить минимальное остовное дерево пошагово. Он начинает с произвольной вершины и последовательно добавляет новые ребра, выбирая на каждом шаге ребро с минимальным весом, которое соединяет уже выбранные вершины с невыбранными.

Вот общий алгоритм решения задачи "Минимальное остовное дерево" с использованием алгоритма Прима:

1. Инициализация:

- Создать пустое множество остовного дерева MST (Minimum Spanning Tree).
- Выбрать произвольную начальную вершину.
- Создать пустое множество посещенных вершин и добавить начальную вершину в это множество.

2. Пока MST не содержит все вершины:
   - Найти все ребра, соединяющие вершины из MST с вершинами, не входящими в MST.
   - Выбрать ребро с минимальным весом из найденных ребер.
   - Добавить выбранное ребро в MST.
   - Добавить вершину, соединенную выбранным ребром, в множество посещенных вершин.
3. Вернуть MST как результат.

Алгоритм Прима можно реализовать с использованием структур данных, таких как очередь с приоритетом (min-heap) для выбора ребер с минимальным весом и хэш-таблица (например, множество) для отслеживания посещенных вершин.

После выполнения алгоритма Прима, MST будет содержать минимальное остовное дерево, которое является подграфом исходного графа и содержит все его вершины. Блок-схема алгоритма прима изображена на рисунке 2.

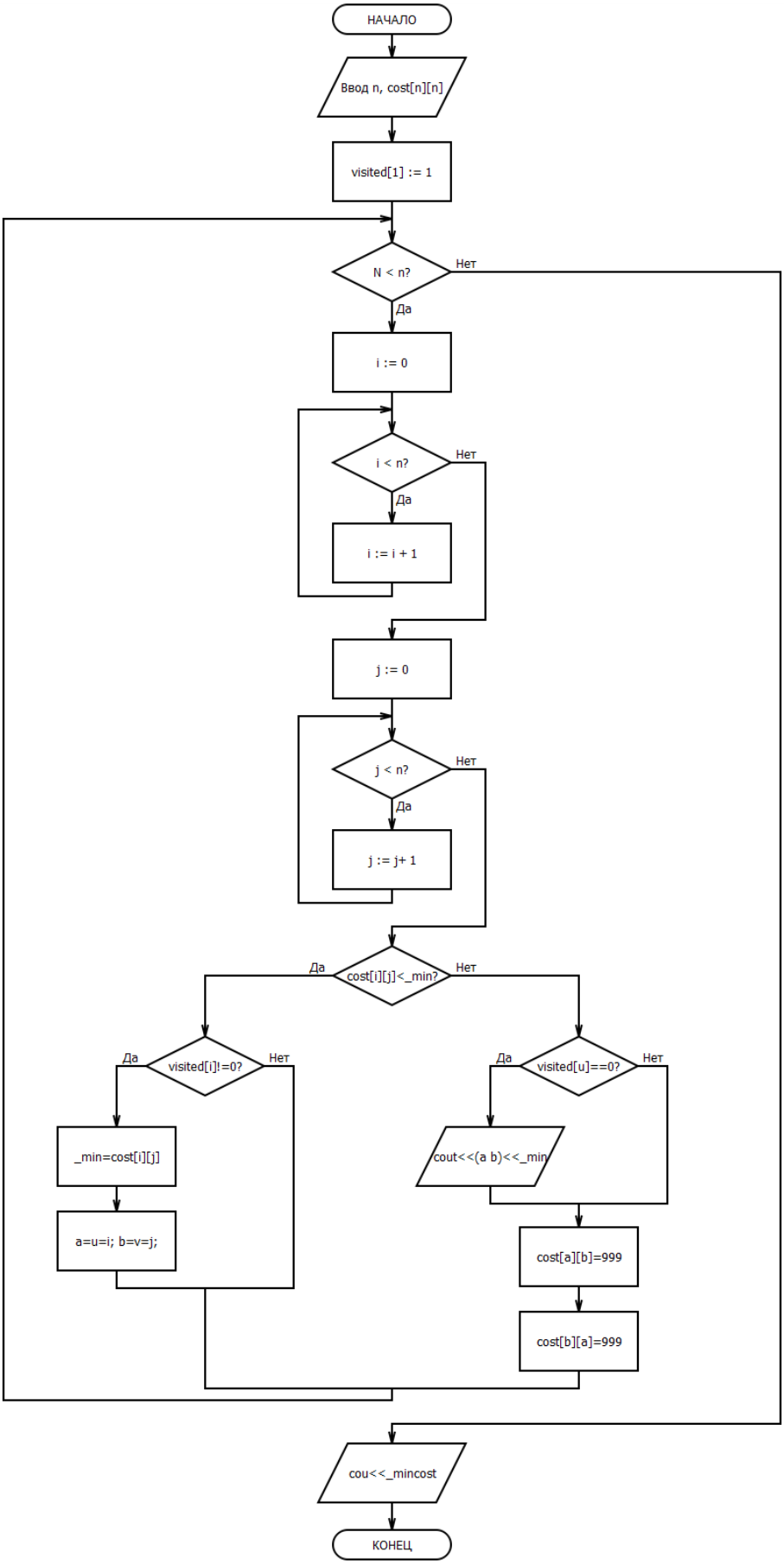

Рисунок 2 – Блок-схема алгоритма Прима

Преобразование инфиксного выражения в постфиксное проводится с помощью следующего алгоритма.

1.3.2 Структуры данных необходимые для решения задачи

Для решения задачи "Минимальное остовное дерево" с использованием алгоритма Прима, требуется использование нескольких структур данных. Ниже описаны основные используемые структуры данных:

1. Граф (визуальное представление на рисунке 3):
    - Граф представляет собой коллекцию вершин и ребер. В контексте задачи "Минимальное остовное дерево" граф является исходным графом, для которого мы ищем минимальное остовное дерево.
    - Граф можно представить с помощью списка смежности или матрицы смежности.

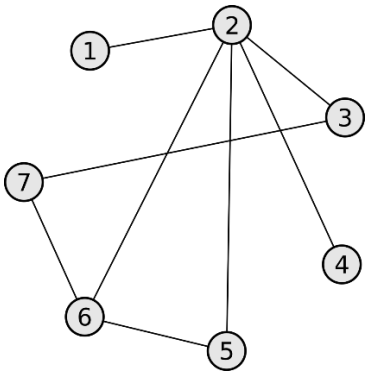

Рисунок 3 – Визуальное представление графа

2. Множество посещенных вершин (графический вид на рисунке 4):
    - Множество посещенных вершин используется для отслеживания вершин, которые уже были добавлены в остовное дерево MST.
    - Можно использовать хэш-таблицу или множество для хранения посещенных вершин и быстрой проверки принадлежности вершины к этому множеству.

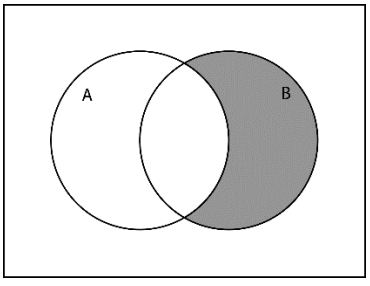

Рисунок 4 – Визуальное представление множества

3. Очередь с приоритетом: (визуальное представление на рисунке 5)
    - Очередь с приоритетом используется для выбора ребер с минимальным весом на каждом шаге алгоритма Прима.
    - Очередь с приоритетом может быть реализована с использованием min-heap, которая обеспечивает эффективное извлечение минимального элемента.

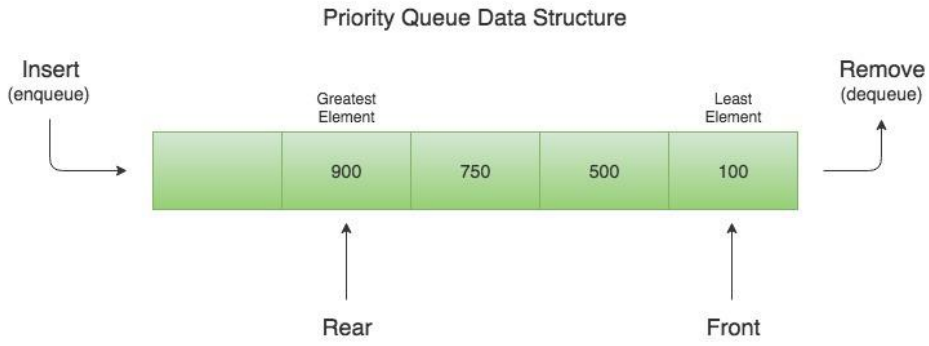

Рисунок 5 – Визуальное представление очереди с приоритетом

Использование этих структур данных позволяет эффективно реализовать алгоритм Прима для поиска минимального остовного дерева.

1.4 Общие требования к программе

Для решения задачи "Минимальное остовное дерево" и реализации алгоритма Прима, программа должна соответствовать следующим общим требованиям:

1. Язык программирования: Программа должна быть написана на языке программирования, поддерживающем нужные структуры данных и операции для работы с графами, таком как C++.
2. Ввод данных: Программа должна предоставлять возможность ввода данных для графа, например, в виде списка ребер с их весами или матрицы смежности. Входные данные должны соответствовать формату, определенному в постановке задачи.
3. Реализация алгоритма Прима: Программа должна содержать реализацию алгоритма Прима для поиска минимального остовного дерева. Алгоритм должен быть правильно реализован с учетом всех его шагов и логики работы.
4. Структуры данных: Программа должна использовать соответствующие структуры данных для представления графа, множества посещенных

вершин и очереди с приоритетом. Реализации структур данных должны быть эффективными и обеспечивать необходимые операции, такие как добавление элементов, удаление минимального элемента, проверка посещенных вершин и другие.

5. Вывод результата: Программа должна выводить результат выполнения алгоритма Прима, то есть минимальное остовное дерево или его представление в нужном формате. Результат должен быть корректным и соответствовать требованиям задачи.
6. Обработка ошибок: Программа должна быть устойчивой к возможным ошибкам во входных данных или некорректным операциям. Она должна обеспечивать обработку ошибок и сообщать пользователю о любых проблемах, возникших в процессе выполнения.
7. Эффективность: Программа должна быть эффективной и обеспечивать выполнение алгоритма Прима за разумное время. Реализации структур данных и алгоритма должны быть оптимизированы для достижения высокой производительности.
8. Документация и комментарии: Программа должна быть хорошо задокументирована и содержать комментарии к ключевым частям кода. Комментарии должны объяснять логику работы, структуры данных и важные алгоритмические шаги.

Общие требования к программе помогут обеспечить правильную и эффективную реализацию алгоритма Прима для поиска минимального остовного дерева.

1.5 Описание структуры программы для решения задачи

Структура программы для решения задачи по поиску минимального остовного дерева в графе может быть следующей:

1. Модуль **Graph**:
    - Класс **Graph** представляет граф и содержит методы для работы с вершинами, связями и весами.
    - Методы включают:

- **addEdge**: добавляет связь между двумя вершинами с заданным весом.
- **getWeight**: возвращает вес между двумя вершинами.
- **getAdjacentVertices**: возвращает список смежных вершин для заданной вершины.
- **getVerticesCount**: возвращает количество вершин в графе.
- **getEdgesCount**: возвращает количество связей в графе.
- **printGraph**: выводит граф на консоль в виде списка вершин и их связей.

2. Модуль **MinimumSpanningTree**:
    - Класс **MinimumSpanningTree** реализует алгоритм поиска минимального остовного дерева.
    - Методы включают:
        - **calculateMST**: выполняет поиск минимального остовного дерева в графе с использованием алгоритма Прима.
        - **getMinimumSpanningTree**: возвращает остовное дерево в виде списка связей.
        - **getTotalWeight**: возвращает вес остовного дерева.
3. Модуль **Main**:
    - Функция **main** является точкой входа в программу.
    - В ней создается объект класса **Graph**, добавляются вершины и связи в граф, а затем создается объект класса **MinimumSpanningTree** и вызывается метод **calculateMST** для поиска минимального остовного дерева.
    - Выводится остовное дерево и его вес на консоль.

Структура программы представляет модульную архитектуру, где каждый модуль отвечает за определенный функционал и имеет свою ответственность. Модули взаимодействуют друг с другом для выполнения задачи поиска минимального остовного дерева в графе.

для взаимодействия с другими модулями. Схема взаимосвязи модулей между собой приведена на рисунке 6.

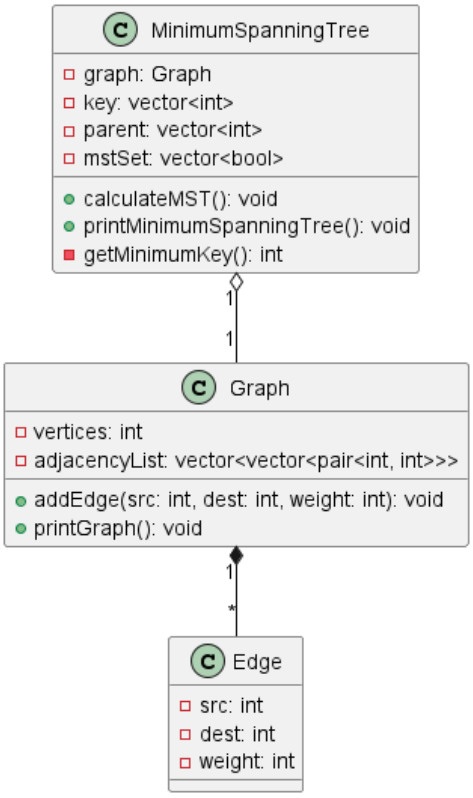

Рисунок 6 – Диаграмма классов проекта минимального остового дерева

1.6 Инструкции по эксплуатации программы

Инструкции по эксплуатации программы для поиска минимального остовного дерева в графе:

1. Сборка проекта:
   - Загрузите исходный код проекта с представленной выше реализацией.
   - Откройте проект в вашей среде разработки, поддерживающей язык C++ (например, Visual Studio, CLion или Code::Blocks).

- Убедитесь, что у вас установлен компилятор C++ и настроены соответствующие компиляторные настройки проекта.
- Соберите проект, чтобы получить исполняемый файл.

2. Запуск программы:
    - После успешной сборки проекта, запустите полученный исполняемый файл.
    - Программа предложит вам выбрать один из пяти тестовых случаев или выполнить тест с пользовательским вводом.
    - В случае выбора тестовых случаев, программа автоматически выполнит алгоритм поиска минимального остовного дерева для каждого теста и выведет результаты на консоль.
    - Если вы выберете пользовательский ввод, вам будет предложено ввести данные о графе, включая количество вершин, связей и их весов.
    - После ввода данных, программа выполнит алгоритм поиска минимального остовного дерева и выведет результаты на консоль.
3. Анализ результатов:
    - После выполнения программы, она выведет остовное дерево графа в виде списка связей и их весов.
    - Также будет выведен общий вес остовного дерева.
    - Вы можете проанализировать результаты для проверки правильности работы алгоритма поиска минимального остовного дерева.
4. Повторное выполнение:
    - Вы можете повторно запускать программу для тестирования других графов или настройки пользовательского ввода.
    - Следуйте инструкциям на экране и вводите соответствующие данные для каждого запуска.

Примечание: убедитесь, что у вас установлены все необходимые компоненты для сборки и запуска программы, а также обратитесь к документации вашей

среды разработки для более подробной информации о настройках и командах компиляции.

1.7 Описание контрольного примера

Тест программы минимального остовного дерева № 1

Исходные данные:

- Количество вершин: 4
- Количество ребер: 5
- Связи и веса (представлены в таблице 1):

Таблица 1 – Исходный граф №1

Вершина	Вершина	Вес
0	1	2
0	2	3
1	2	1
1	3	4

Выходные данные:

- Остовное дерево (представлено в таблице 2):

Таблица 2 – Представление остовного дерева №1

Вершина	Вершина	Вес
0	1	2
1	2	1
1	3	4

- Вес остовного дерева: 7

Результат работы программы представлен на рисунке 7:

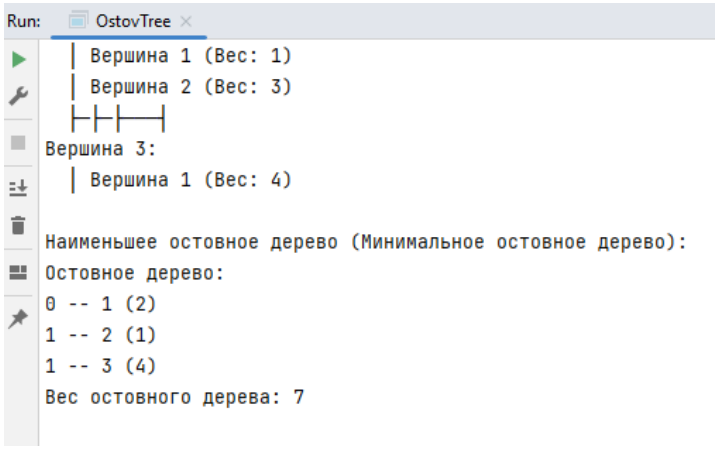

Рисунок 7 – Тест программы № 1

Тест программы минимального остовного дерева № 2

Исходные данные:

- Количество вершин: 6
- Количество ребер: 9
- Связи и веса (представлены в таблице 3):

Таблица 3 - Исходный граф №2

Вершина	Вершина	Вес
0	1	2
0	2	3
1	2	1
1	3	4
1	5	3
2	3	4
2	4	2
3	5	4
4	5	2

Выходные данные:

- Остовное дерево(представлено в таблице 4):
  Таблица 4 – Представление остовного дерева №2

Вершина	Вершина	Вес
0	1	2
1	2	1
1	3	4
5	4	4
1	5	3

- Вес остовного дерева: 14

Результат работы программы представлен на рисунке 8:

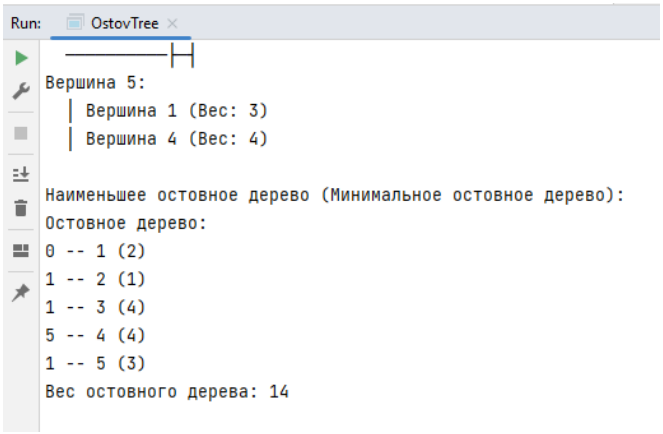

Рисунок 8 – Тест программы № 2

Тест программы минимального остовного дерева № 3

Исходные данные:

- Количество вершин: 6
- Количество ребер: 9
- Связи и веса (представлены в таблице 5)::

Таблица 5 - Исходный граф №3

Вершина	Вершина	Вес
0	1	1
0	2	5
1	3	3
1	4	6
2	3	4
2	4	2
3	5	4
4	5	2

Выходные данные:

Остовное дерево(представлено в таблице 6):

Таблица 6 – Представление остовного дерева №3

Вершина	Вершина	Вес
0	1	1
3	2	4
1	3	3
2	4	2
4	5	2

- Вес остовного дерева: 12

Результат работы программы представлен на рисунке 9:

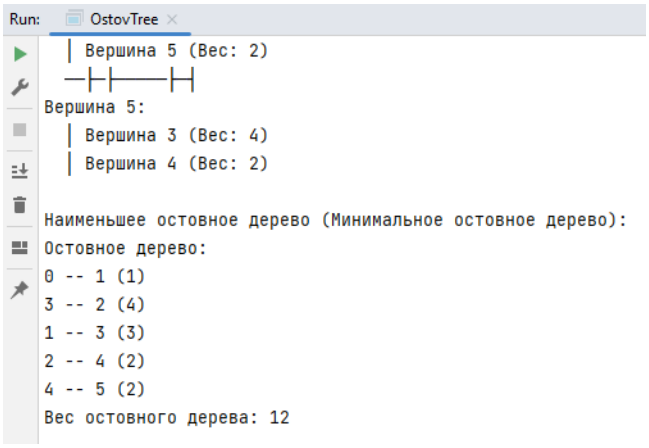

Рисунок 9 – Тест программы № 3

Тест программы минимального остовного дерева № 4

Исходные данные:

- Количество вершин: 6
- Количество ребер: 8
- Связи и веса (представлены в таблице 8):
  Таблица 7 - Исходный граф №4

Вершина	Вершина	Вес
0	1	2
0	2	3
1	2	1
1	3	4
1	5	3
2	3	4
2	4	2
3	5	4
4	5	2

Выходные данные:

- Остовное дерево (представлено в таблице 8):
  Таблица 8 - Представление остовного дерева №4

Вершина	Вершина	Вес
0	1	2

1	2	1
1	3	4
5	4	4
1	5	3

- Вес остовного дерева: 14

Результат работы программы представлен на рисунке 10:

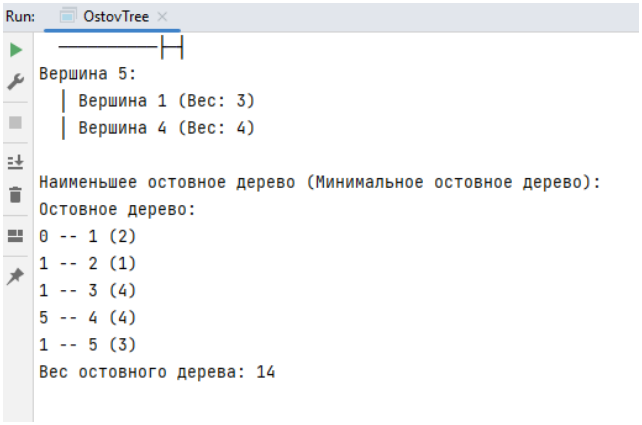

Рисунок 10 – Тест программы № 4

2 Графы: нахождение предков в деревьях и наименьших общих предков

2.1 Постановка задачи

2.1.1 Нахождение предков в деревьях:

- Дано дерево с корнем и набор запросов вида (v, u), где v и u - вершины дерева.
- Задача состоит в том, чтобы найти все предки вершины v, которые находятся на пути от корня до вершины u.
- Предки - это вершины, через которые проходит путь от корня до заданной вершины.
- Обычно для решения этой задачи используются алгоритмы обхода дерева, такие как обход в глубину (DFS) или обход в ширину (BFS).

- Результатом является список всех предков вершины v.

2.1.2 Наименьший общий предок (Lowest Common Ancestor - LCA):

- Дано дерево с корнем и набор запросов вида (v, u), где v и u - вершины дерева.
- Задача состоит в том, чтобы найти наименьшего общего предка (общую вершину) для пары вершин v и u.
- Наименьший общий предок - это вершина, которая является ближайшим общим предком для v и u.
- Обычно для решения этой задачи используются алгоритмы на основе предварительной обработки дерева, такие как алгоритм Двоичного Подъема (Binary Lifting) или алгоритм Эйлерова обхода.
- Результатом является наименьший общий предок для каждого запроса (v, u).

В обоих задачах ключевым является понятие предков - вершин, через которые проходят пути от корня до заданных вершин. Нахождение предков в деревьях полезно для анализа и обработки структуры дерева, а наименьший общий предок позволяет находить общие свойства и отношения между вершинами в дереве.

Решение этих задач требует использования алгоритмов и структур данных, специально разработанных для работы с графами и деревьями. Например, для нахождения предков в деревьях может использоваться массив предков или таблица предков, а для наименьшего общего предка - дерево отрезков или метод двоичного подъема.

2.2 Описание входной и выходной информации

2.2.1 Входная информация:

- Для задачи нахождения предков в деревьях обычно требуется предоставить следующую информацию:
    - Дерево с корнем: указание вершин и связей между ними. Это может быть представлено в виде списка вершин с указанием их родителей или дочерних вершин.
    - Запросы: список запросов вида (v, u), где v и u - вершины дерева, для которых нужно найти предков.
- Для задачи наименьшего общего предка требуется предоставить следующую информацию:
    - Дерево с корнем: указание вершин и связей между ними, так же как и для задачи нахождения предков в деревьях.
        - Запросы: список запросов вида (v, u), где v и u - вершины дерева, для которых нужно найти наименьшего общего предка.

2.2.2 Выходная информация:

- Для задачи нахождения предков в деревьях ожидается следующий формат выходных данных:
    - Для каждого запроса (v, u) необходимо предоставить список предков вершины v на пути от корня до вершины u.
- Для задачи наименьшего общего предка ожидается следующий формат выходных данных:
    - Для каждого запроса (v, u) необходимо предоставить наименьшего общего предка (общую вершину) для пары вершин v и u.

Выходная информация должна быть представлена в понятном и удобочитаемом формате, который позволяет легко интерпретировать

результаты и использовать их для дальнейшего анализа или обработки данных.

Правильное предоставление входных данных и ожидаемого формата выходных данных играет важную роль в эффективном решении задач нахождения предков в деревьях и наименьшего общего предка, так как это определяет правильное взаимодействие с алгоритмами и обеспечивает корректность и точность результатов.

2.3 Алгоритм решения задачи

2.3.1 Алгоритм решения задачи нахождения предков в деревьях:

1. Создание структуры данных для хранения дерева с корнем, например, используя списки смежности или указатели на родителей.
2. Обход дерева с корнем с помощью обхода в глубину (DFS) или обхода в ширину (BFS).
3. При обходе каждой вершины дерева сохранение информации о её родителях.
4. Для каждого запроса (v, u) нахождение пути от вершины v до вершины u, используя сохраненную информацию о родителях.
5. Возврат списка предков вершины v на пути к вершине u в качестве результата для каждого запроса.

2.3.2 Алгоритм решения задачи наименьшего общего предка:

1. Создание структуры данных для хранения дерева с корнем, так же как и для задачи нахождения предков в деревьях.

2. Обход дерева с корнем с помощью обхода в глубину (DFS) или обхода в ширину (BFS).
3. При обходе каждой вершины дерева сохранение информации о её родителях.
4. Для каждого запроса (v, u) нахождение пути от вершины v до вершины u, используя сохраненную информацию о родителях.
5. Нахождение наименьшего общего предка для вершин v и u на найденном пути.
6. Возврат наименьшего общего предка в качестве результата для каждого запроса.

Блок-схема алгоритмов представленна на рисунке 11.

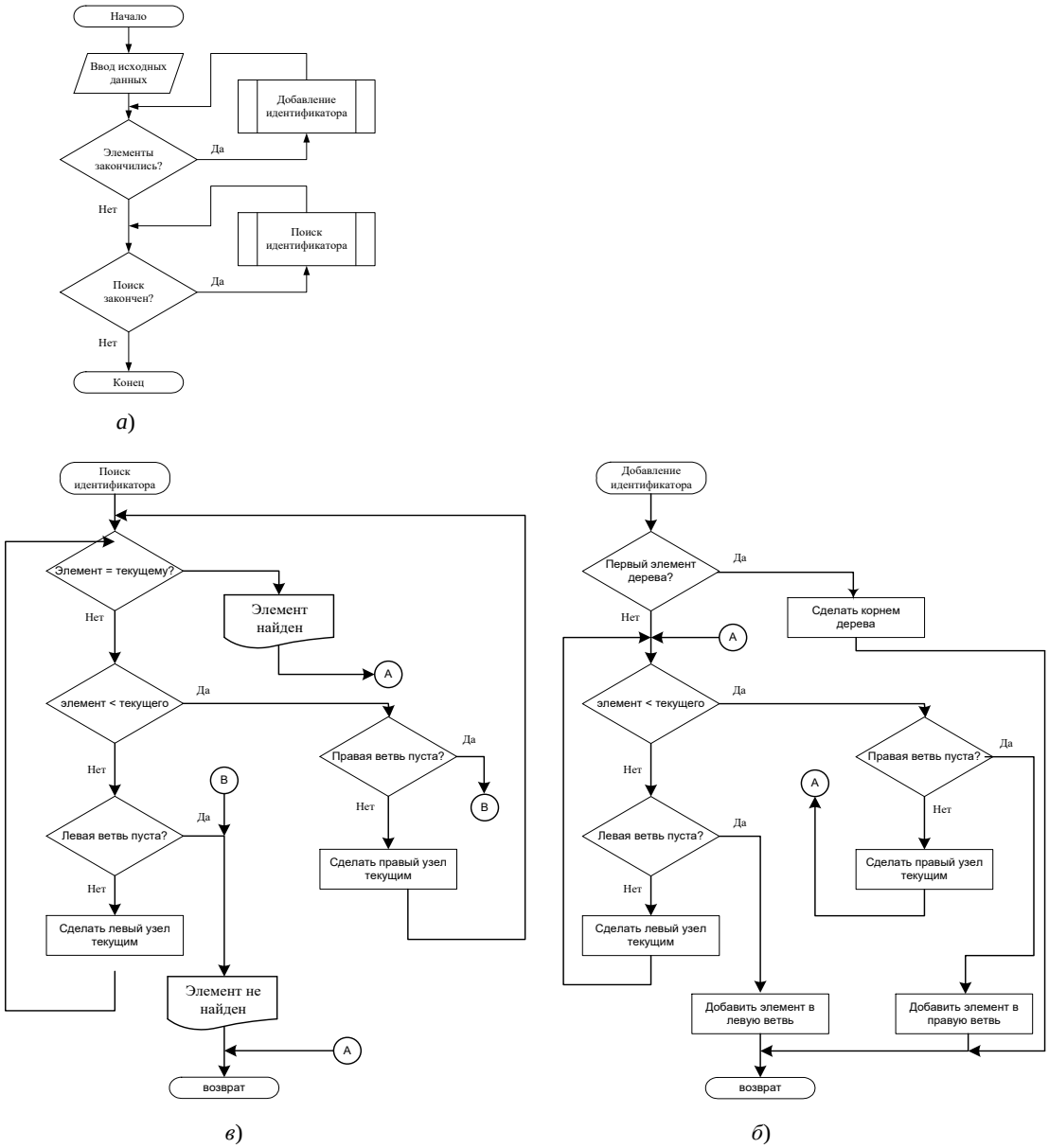

Рисунок 11 - Блок-схема нахождение предков в деревьях и наименьших общих предков

2.4 Общие требования к программе

1. Программа должна принимать входные данные, которые представляют собой:

    - Граф, представленный в виде списка смежности или матрицы смежности.

- Запросы нахождения предков и наименьшего общего предка в формате (v, u), где v и u - вершины графа.

2. Программа должна реализовывать алгоритмы нахождения предков в деревьях и наименьшего общего предка, описанные в пункте 2.3.
3. Программа должна выводить результаты для каждого запроса, представленные в читаемом формате:
    - Для запроса нахождения предков: список предков вершины u от вершины v.
    - Для запроса нахождения наименьшего общего предка: наименьший общий предок вершин v и u.
4. Программа должна обрабатывать случаи некорректных входных данных и выводить соответствующие сообщения об ошибке.
5. Программа должна быть эффективной с точки зрения времени выполнения и использования памяти, особенно при работе с большими деревьями и большим количеством запросов.
6. Программа должна быть написана с использованием подходящего языка программирования и соответствовать принятому стандарту кодирования.
7. Программа должна содержать комментарии и объяснения к коду, чтобы облегчить понимание алгоритмов и логики программы.
8. Программа должна проходить проверку на различных тестовых случаях и возвращать правильные результаты для каждого запроса.
9. Программа должна быть гибкой и модульной, позволяя легко расширять функциональность или внедрять ее в другие проекты.

10. Программа должна быть достаточно документирована, включая описание входных и выходных данных, описание алгоритмов и требования к программе.

2.5. Описание структуры программы

Структура программы для задачи нахождения предков в деревьях и наименьших общих предков может быть описана следующим образом:

1. **Граф (Graph):**
    - Класс, представляющий граф.
    - Содержит приватные члены:
        - **vertices** - количество вершин в графе.
        - **adjacencyList** - список смежности графа, представленный вектором векторов пар **pair<int, int>**, где первое значение в паре - номер смежной вершины, а второе значение - вес ребра.
    - Содержит публичные методы:
        - **addEdge(src, dest, weight)** - добавляет ребро между вершинами **src** и **dest** с весом **weight** в граф.
        - **printGraph()** - выводит информацию о графе, включая вершины и связи с весами.
2. **Наименьший общий предок (LCA):**
    - Класс, представляющий наименьшего общего предка.
    - Принимает объект графа в конструкторе.
    - Содержит приватные члены:
        - **graph** - ссылка на объект графа.
        - **parent** - вектор, хранящий предков каждой вершины.
        - **depth** - вектор, хранящий глубину каждой вершины.
    - Содержит приватные методы:

- **dfs(vertex, par, dep)** - рекурсивная функция обхода графа в глубину для вычисления предков и глубины каждой вершины.
- **findParent(vertex, levelDiff)** - находит предка вершины на заданном уровне.

- Содержит публичные методы:
  - **printAncestors(vertex)** - выводит предков заданной вершины.
  - **findLCA(u, v)** - находит наименьшего общего предка для заданных вершин **u** и **v**.

3. **Функция main:**
   - Тестирование программы нахождения предков в деревьях и наименьших общих предков.
   - Создание объекта графа и добавление ребер.
   - Создание объекта наименьшего общего предка с передачей объекта графа в конструктор.
   - Вызов методов наименьшего общего предка для получения результатов и вывода на экран.

Структура программы позволяет организовать работу с графом и нахождение предков в деревьях, а также наименьшего общего предка для заданных вершин. Классы **Graph** и **LCA** являются независимыми компонентами, что обеспечивает удобство использования и модульность кода.

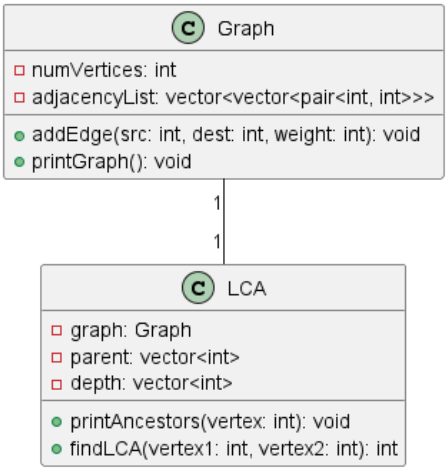

Рисунок 12 – Диаграмма классов проекта двусвязного списка

2.6 Инструкция по эксплуатации программы

Инструкция по эксплуатации программы для нахождения предков в деревьях и наименьших общих предков:

1. **Компиляция программы:**
   - Убедитесь, что у вас установлен компилятор C++ (например, g++).
   - Скачайте исходный код программы.
   - Откройте командную строку (терминал) и перейдите в папку с исходным кодом программы.
   - Выполните следующую команду для компиляции программы:

```
g++ -o main main.cpp Graph.cpp LCA.cpp
```

   - После успешной компиляции будет создан исполняемый файл **main** (или **main.exe** в Windows).
2. **Запуск программы:**
   - Запустите исполняемый файл **main** (или **main.exe** в Windows) в командной строке или двойным щелчком мыши.
   - Программа начнет выполнение и выведет результаты на экран.

3. **Использование программы:**

    - При запуске программы уже заданы несколько тестовых наборов данных.
    - Результаты каждого набора данных будут выводиться на экран с соответствующими пояснениями на русском языке.
    - Вы также можете добавить свои собственные тестовые наборы данных, изменив код в функции **runTests()** в файле **main.cpp**.
    - Для каждого тестового набора данных программа выводит графическое представление дерева, список предков заданной вершины и наименьшего общего предка для заданных вершин.
    - При необходимости вы можете изменить код программы для ввода собственных данных или проведения дополнительных тестов.

4. **Выход из программы:**

    - После завершения работы программы вы можете закрыть окно командной строки или нажать клавишу **Ctrl+C**.

Программа предоставляет простой и интуитивно понятный способ нахождения предков в деревьях и наименьших общих предков. Следуйте инструкции по эксплуатации для использования программы и получения результатов.

.

2.7 Описание контрольного примера

Тестовый набор данных № 1:

- Входные данные:
    - Граф (представлен в таблице 9):

Таблица 9 -Исходный граф набора данных № 1

Вершина	Вершина
0	1
0	2
1	0
1	3
2	0
2	4
2	5
3	1
4	2
5	2

- Выходные данные:
  - Предки вершины 3: 3 1 0
  - Наименьший общий предок для вершин 3 и 5: 0

Результат работы программы с тестовым набором данных № 1 представлен на рисунке 13.

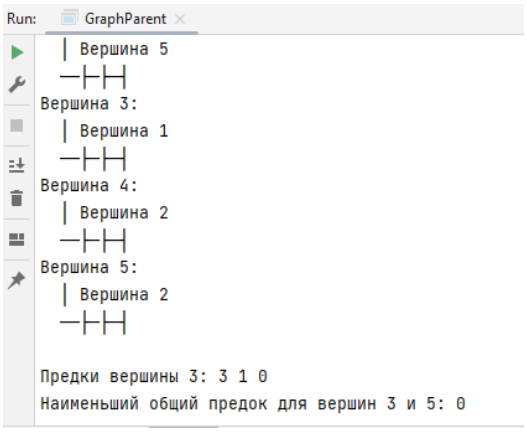

Рисунок – 13 Тестовый пример №1, для задачи №2

Тестовый набор данных № 2:

- Входные данные:
  - Граф(представлен в таблице 10):

Таблица 10 - Исходный граф набора данных № 2

Вершина	Вершина
0	1
1	0
1	2
2	1

2	3
3	2
3	4
4	3

- Выходные данные:
  - Предки вершины 4: 4 3 2 1 0
  - Наименьший общий предок для вершин 4 и 2: 2

Результат работы программы с тестовым набором данных № 2 представлен на рисунке 14.

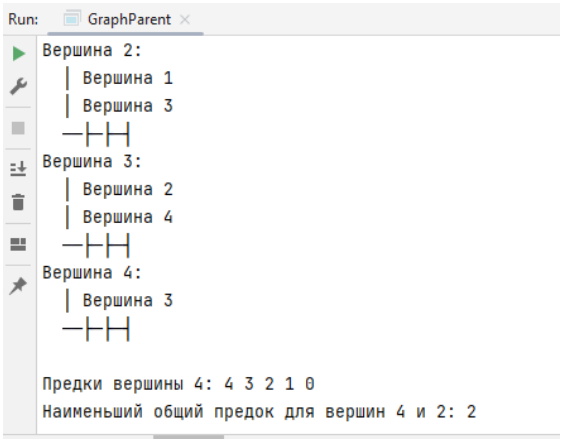

Рисунок – 14 Тестовый пример №2, для задачи №2

Тестовый набор данных № 3:

- Входные данные:
  - Граф (представлен в таблице 11):

Таблица 11 - Исходный граф набора данных № 3

Вершина	Вершина
0	1
0	2
1	0
1	3
1	4
2	0
2	5
2	6
3	1
5	2
6	2
6	7
7	6

- Выходные данные:
  - Предки вершины 3: 3 1 0
  - Предки вершины 5: 5 2 0
  - Наименьший общий предок для вершин 3 и 5: 0

Результат работы программы с тестовым набором данных № 3 представлен на рисунке 15.

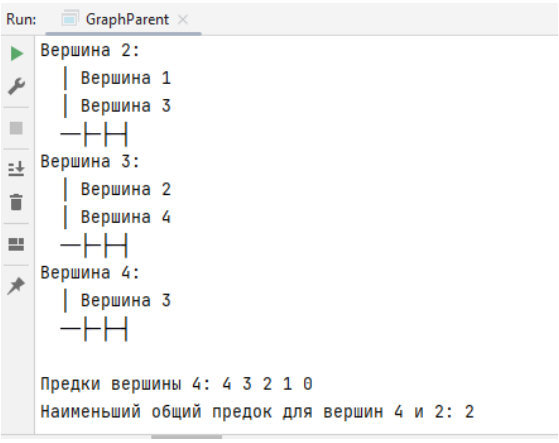

Рисунок – 15 Тестовый пример №3, для задачи №2

Заключение

В данной программе реализован алгоритм нахождения предков в деревьях и наименьших общих предков. Она позволяет эффективно работать с деревьями и определять отношения между вершинами. Программа предоставляет графическое представление дерева, список предков для заданной вершины и наименьший общий предок для заданных вершин.

Основные функции программы:

- Конструирование графа: пользователь может вводить данные о вершинах и их связях.
- Нахождение предков: программа выводит список предков для заданной вершины.
- Нахождение наименьшего общего предка: программа находит наименьший общий предок для двух заданных вершин.

Программа имеет простой интерфейс, который позволяет легко использовать ее как для предопределенных тестовых наборов данных, так и для пользовательского ввода. Она предоставляет понятные результаты на русском языке, что упрощает восприятие полученных данных.

Программа может быть полезна во многих областях, где требуется работа с деревьями и анализ их структуры. Это может включать программирование алгоритмов на графах, биоинформатику, анализ данных и другие приложения, где необходимо определить отношения между элементами дерева.

В заключение, программа нахождения предков в деревьях и наименьших общих предков предоставляет эффективный и простой способ работы с деревьями. Она является полезным инструментом для анализа структуры деревьев и определения отношений между вершинами.

Библиографический список

1. Гасфилев, В. М. (2007). Структуры данных и алгоритмы в C++. БХВ-Петербург.
2. Weiss, M. A. (2013). Data Structures and Algorithm Analysis in C++. Pearson.
3. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th Edition). Addison-Wesley Professional.
4. Goodrich, M. T., Tamassia, R., & Mount, D. M. (2011). Data Structures and Algorithms in C++. Wiley.
5. Алексеев, И. В. (2015). Структуры данных и алгоритмы: учебник для вузов. Москва: Бином.

ПРИЛОЖЕНИЕ А

Исходный код Graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include <vector>

struct Edge {
    int src;
    int dest;
    int weight;
};

class Graph {
public:
    Graph(int numVertices);
    void addEdge(int src, int dest, int weight);
    std::vector<Edge> getEdges();
    int numVertices;
    std::vector<std::vector<int>> adjacencyMatrix;
    void printGraph();
};

#endif
```

Исходный код Graph.cpp

```
#include "include/Graph.h"
#include <iostream>
Graph::Graph(int numVertices)
{
```

```
    this->numVertices = numVertices;
    adjacencyMatrix.resize(numVertices, std::vector<int>(numVertices,
0));
}

void Graph::addEdge(int src, int dest, int weight)
{
    adjacencyMatrix[src][dest] = weight;
    adjacencyMatrix[dest][src] = weight;
}

std::vector<Edge> Graph::getEdges()
{
    std::vector<Edge> edges;
    for (int i = 0; i < numVertices; ++i)
    {
        for (int j = i + 1; j < numVertices; ++j)
        {
            if (adjacencyMatrix[i][j] != 0)
            {
                Edge edge;
                edge.src = i;
                edge.dest = j;
                edge.weight = adjacencyMatrix[i][j];
                edges.push_back(edge);
            }
        }
    }
    return edges;
}

void Graph::printGraph()
{
```

```
    std::cout << "Отображение графа:\n";

    for (int i = 0; i < numVertices; ++i)
    {
        std::cout << "Вершина " << i << ":\n";

        // Вывод связей
        for (int j = 0; j < numVertices; ++j)
        {
            if (adjacencyMatrix[i][j] != 0)
            {
                std::cout << "  ";
                std::cout << "│ "; // Вертикальная линия
                std::cout << "Вершина " << j << " (Вес: " <<
adjacencyMatrix[i][j] << ")\n";
            }
        }

        // Вывод горизонтальных линий
        if (i != numVertices - 1)
        {
            std::cout << "  ";
            for (int j = 0; j < numVertices; ++j)
            {
                if (adjacencyMatrix[i][j] != 0)
                {
                    std::cout << "├─"; // Горизонтальная линия
                }
                else
                {
                    std::cout << "──"; // Пустое место
                }
            }
```

```
            std::cout << "┤\n";
        }
    }
}
```

Исходный код MinimumSpanningTree.h

```
#ifndef MINIMUMSPANNINGTREE_H
#define MINIMUMSPANNINGTREE_H

#include "Graph.h"
#include "PriorityQueue.h"

class MinimumSpanningTree {
public:
    MinimumSpanningTree(Graph& graph);
    std::vector<Edge> getMinimumSpanningTree();

private:
    Graph& graph;
    PriorityQueue priorityQueue;
    std::vector<int> key;
    std::vector<int> parent;
    std::vector<bool> inMST;

    void initialize();
    void primAlgorithm(int startVertex);
};

#endif
```

Исходный код MinimumSpanningTree.cpp

```
#include "include/MinimumSpanningTree.h"
#include <climits>

MinimumSpanningTree::MinimumSpanningTree(Graph& graph) : graph(graph),
priorityQueue(graph.numVertices) {
    initialize();
}

std::vector<Edge> MinimumSpanningTree::getMinimumSpanningTree() {
    std::vector<Edge> minimumSpanningTree;

    for (int i = 0; i < graph.numVertices; ++i) {
        if (!inMST[i]) {
            primAlgorithm(i);
        }
    }

    for (int i = 1; i < graph.numVertices; ++i) {
        Edge edge;
        edge.src = parent[i];
        edge.dest = i;
        edge.weight = graph.adjacencyMatrix[i][parent[i]];
        minimumSpanningTree.push_back(edge);
    }

    return minimumSpanningTree;
}

void MinimumSpanningTree::initialize() {
    key.resize(graph.numVertices, INT_MAX);
    parent.resize(graph.numVertices, -1);
```

```
    inMST.resize(graph.numVertices, false);
}

void MinimumSpanningTree::primAlgorithm(int startVertex) {
    priorityQueue.insert(startVertex, 0);
    key[startVertex] = 0;

    while (!priorityQueue.isEmpty()) {
        int currentVertex = priorityQueue.extractMin();
        inMST[currentVertex] = true;

        for (int i = 0; i < graph.numVertices; ++i) {
            int weight = graph.adjacencyMatrix[currentVertex][i];

            if (weight != 0 && !inMST[i] && weight < key[i]) {
                priorityQueue.insert(i, weight);
                key[i] = weight;
                parent[i] = currentVertex;
            }
        }
    }
}
```

Исходный код PriorityQueue.h

```
#ifndef PRIORITYQUEUE_H
#define PRIORITYQUEUE_H

#include <vector>

struct HeapNode {
```

```
    int vertex;
    int key;
};

class PriorityQueue {
public:
    PriorityQueue(int capacity);
    bool isEmpty();
    void insert(int vertex, int key);
    int extractMin();

private:
    std::vector<HeapNode> heap;
    std::vector<int> position;
    int capacity;
    int size;

    void minHeapify(int index);
    void swapHeapNodes(int index1, int index2);
    int parent(int index);
    int leftChild(int index);
    int rightChild(int index);
};

#endif
```

Исходный код PriorityQueue.cpp

```
#include "include/PriorityQueue.h"
PriorityQueue::PriorityQueue(int capacity) {
    this->capacity = capacity;
    size = 0;
    heap.resize(capacity);
```

```
    position.resize(capacity);
}

bool PriorityQueue::isEmpty() {
    return size == 0;
}

void PriorityQueue::insert(int vertex, int key) {
    if (size == capacity) {
        return;
    }

    HeapNode newNode;
    newNode.vertex = vertex;
    newNode.key = key;
    heap[size] = newNode;
    position[vertex] = size;

    int current = size;
    int parentIndex = parent(current);
    while (current != 0 && heap[current].key < heap[parentIndex].key)
{
        swapHeapNodes(current, parentIndex);
        current = parentIndex;
        parentIndex = parent(current);
    }

    ++size;
}

int PriorityQueue::extractMin() {
```

```
    if (isEmpty()) {
        return -1;
    }

    HeapNode minNode = heap[0];
    HeapNode lastNode = heap[size - 1];
    heap[0] = lastNode;
    position[minNode.vertex] = -1;
    position[lastNode.vertex] = 0;

    --size;
    minHeapify(0);

    return minNode.vertex;
}

void PriorityQueue::minHeapify(int index) {
    int smallest = index;
    int leftChildIndex = leftChild(index);
    int rightChildIndex = rightChild(index);

    if (leftChildIndex < size && heap[leftChildIndex].key <
heap[smallest].key) {
        smallest = leftChildIndex;
    }

    if (rightChildIndex < size && heap[rightChildIndex].key <
heap[smallest].key) {
        smallest = rightChildIndex;
    }

    if (smallest != index) {
```

```
        swapHeapNodes(index, smallest);
        minHeapify(smallest);
    }
}

void PriorityQueue::swapHeapNodes(int index1, int index2) {
    HeapNode temp = heap[index1];
    heap[index1] = heap[index2];
    heap[index2] = temp;

    position[heap[index1].vertex] = index1;
    position[heap[index2].vertex] = index2;
}

int PriorityQueue::parent(int index) {
    return (index - 1) / 2;
}

int PriorityQueue::leftChild(int index) {
    return 2 * index + 1;
}

int PriorityQueue::rightChild(int index) {
    return 2 * index + 2;
}
```

ПРИЛОЖЕНИЕ Б

Исходный код Graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include <vector>

class Graph {
public:
    explicit Graph(int numVertices);

    void addEdge(int u, int v);
    void printGraph() const;

    int getNumVertices() const;
    const std::vector<int>& getAdjacentVertices(int vertex) const;

private:
    int numVertices;
    std::vector<std::vector<int>> adjacencyList;
};

#endif
```

Исходный код Graph.cpp

```
 #include "include/Graph.h"
#include <iostream>

Graph::Graph(int numVertices) : numVertices(numVertices) {
    adjacencyList.resize(numVertices);
}

void Graph::addEdge(int u, int v) {
    adjacencyList[u].push_back(v);
    adjacencyList[v].push_back(u);
}

void Graph::printGraph() const {
    for (int i = 0; i < numVertices; i++) {
        std::cout << "Вершина " << i << ":\n";
        for (const auto& vertex : adjacencyList[i]) {
            std::cout << "  │ Вершина " << vertex << "\n";
        }
        std::cout << "  ──┼─┼─┤\n";
    }
}
```

```
int Graph::getNumVertices() const {
    return numVertices;
}

const std::vector<int>& Graph::getAdjacentVertices(int vertex) const {
    return adjacencyList[vertex];
}
```

Исходный код LCA.h

```
#ifndef LCA_H
#define LCA_H

#include "Graph.h"

class LCA {
public:
explicit LCA(const Graph& graph);

void printAncestors(int vertex) const;
int findLCA(int u, int v) const;

private:
const Graph& graph;
std::vector<int> parent;
std::vector<int> depth;

void dfs(int vertex, int parent, int depth);
int findParent(int vertex, int levelDiff) const;
};

#endif
```

Исходный код LCA.cpp

```
#include "include/LCA.h"
#include <iostream>
LCA::LCA(const Graph& graph) : graph(graph) {
    int numVertices = graph.getNumVertices();
    parent.resize(numVertices);
    depth.resize(numVertices);

    dfs(0, -1, 0);
}

void LCA::dfs(int vertex, int par, int dep) {
    parent[vertex] = par;
    depth[vertex] = dep;
```

```
    const std::vector<int>& adjacentVertices =
graph.getAdjacentVertices(vertex);
    for (int adjVertex : adjacentVertices) {
        if (adjVertex != par) {
            dfs(adjVertex, vertex, dep + 1);
        }
    }
}

void LCA::printAncestors(int vertex) const {
    while (vertex != -1) {
        std::cout << vertex << " ";
        vertex = parent[vertex];
    }
    std::cout << std::endl;
}
int LCA::findParent(int vertex, int levelDiff) const {
    while (levelDiff > 0) {
        vertex = parent[vertex];
        levelDiff--;
    }
    return vertex;
}
int LCA::findLCA(int u, int v) const {
    int uDepth = depth[u];
    int vDepth = depth[v];

    if (uDepth < vDepth) {
        v = findParent(v, vDepth - uDepth);
    } else if (vDepth < uDepth) {
        u = findParent(u, uDepth - vDepth);
    }
    while (u != v) {
        u = parent[u];
        v = parent[v];
    }

    return u;
}
```